mcobject®
precision data management

eXtremeDB, the real-time embedded database for devices that are eXtremely innovative.

eXtremeDB®

# Getting Started - Core
# Version 6.0

33309 1st Way S., A-208
Federal Way, WA 98003 USA

Phone: +1-425-888-8505
Fax: +1-253-878-5481
E-mail: info@mcobject.com
www.mcobject.com

October 28, 2014

## Table of Contents

## Introduction

*e*X*treme*DB is a **development tool** implemented as a highly optimized set of C libraries, and C# and Java APIs, that database-enables your C/C++, C# or Java application, providing high performance database functionality. The following tutorial is intended to help you quickly begin putting the power of *e*X*treme*DB to work, through a series of sample applications designed to introduce one concept or feature at a time. Often one or more of these sample applications can serve as a model or starting point for a more specialized application that addresses your specific needs.

Note that this document focuses on the core *e*X*treme*DB API and accompanies the *e*X*treme*DB *User Guide* and *Reference Guide*, which should be read and referenced frequently to broaden your experience with this core functionality. It also supplements the *e*X*treme*DB *release_notes* for platform-specific details and *Samples* for brief explanations of each sample application. If you have purchased one of the special editions, such as *e*X*treme*DB *Financial Edition* (FE), *e*X*treme*SQL (SQL), *e*X*treme*DB *High Availability* (HA), *Transaction Logging* (TL) or *Cluster* (CL), this tutorial will still help you to understand the *e*X*treme*DB core functionality, while it will be of further help to read the *User Guide*, *Reference Guide* and *Getting Started* documents for that special edition.

A further note to developers using the *e*X*treme*DB C# or Java Native Interface: though your applications might not need to directly interface with the core runtime functions, it is recommended that you read the sections describing the C/C++ API to gain a deeper understanding of the underlying database runtime implementation.

## Installation

The *eXtreme*DB package you purchased is most often delivered (downloaded) as a tar archive on Unix-hosted systems or as an installation executable on Windows-hosted systems. Simply extract the contents of this archive to a convenient directory. The directory tree will look something like this:

```
c:\McObject\Fusion\win32vs2010\eXtremeDB
        docs            (product documentation)
        host            (host development tools mcocomp and mcorcomp)
        include         (C source code header files)
        samples         (sample applications)
        target          (target system binaries )
```

The samples directory contains dozens of individual samples that demonstrate important *eXtreme*DB features. Each sample is provided with its own makefile for UNIX systems, or a host platform project file for your development platform (Microsoft Visual Studio, WindRiver Tornado, GreenHills Multi project files, etc.), and complete sample source code and schema files.  Building and running these samples in the debugger on your host development system is a quick way to learn about the individual *eXtreme*DB features and API functions. Note that the "core" samples are provided in the subdirectories under /samples/core.

## Basic concepts

Before you begin using *eXtreme*DB, it’s important to understand some basic concepts:

- The *eXtreme*DB database runtime is implemented as a set of C-language libraries that are linked into your application. *eXtreme*DB is not a “database server” application but rather a fully embedded database engine, also sometimes referred to as an “in-process” database system. Linking the *eXtreme*DB runtime with your application adds database capabilities to it -- the application becomes a light-weight “database server“ itself.
- *eXtreme*DB development is *cross-platform*: the application is developed on one system (the *host* system) and run on another (the *target* system). For example it is common to develop an *eXtreme*DB-based application on an x86 Linux platform and run it on a Linux-based PowerPC target. Sometimes the host and the target platforms are the same (for example Microsoft Windows applications, or QNX x86-based applications).
- The core *eXtreme*DB product provides data access through the ‘native’ C/ C++, C#/.NET (on Windows platforms) and Java APIs. In addition to the C/C++, C# and Java APIs, an SQL API (*eXtreme*SQL) is available.
- Database development using the core *eXtreme*DB API consists of three stages on the host environment:
    1. Data layout definition (creating what is referred to as a *database schema*) and compilation (which produces database interface files, including the *database dictionary*; the database dictionary is a binary representation of the database schema that the *eXtreme*DB run-time uses to understand the data layout, which fields participate in which indexes, etc.).
    2. Using the core *eXtreme*DB API and database interfaces in the application.
    3. Compiling the database interface files with the application source code and linking the application with *eXtreme*DB runtime libraries.
- For C/C++ applications:
    1. C/C++ applications create the database dictionary at *compile* time by compiling the database schema.
    2. The database schema is defined in a high-level C++ -like Data Definition Language (DDL) and then compiled to produce the database dictionary and database access API in the form of a .h header file and a .c implementation file.
    3. The database access API is different for different databases and is generated based on the content of the DDL file.
    4. With simple DDL declarations and/or compiler options, you create all-in-memory databases, “persistent” databases, or hybrid databases that combine both “transient” and “persistent” classes.
- For C# (.NET) applications:
    1. .NET applications define the database dictionary at *runtime*.
    2. Database objects are defined as C# classes; attributes such as “`[Persistent]` ” and “ `[Indexable]` ” are used to declare specific database-related properties for these classes and data elements.

    3. These attributes are examined through reflection at run-time to generate the database dictionary.
    4. Using the `ExtremeDB` namespace enables applications to import class definitions such as *Database, Connection* and *Cursor* and use them to perform database operations.
    5. The interface to the *e*X*treme*DB runtime libraries is implemented through a set of dynamic link libraries located in the subdirectory target/bin.so.

- For Java applications:
    1. Java applications define the database dictionary at *runtime*.
    2. Database objects are defined as Java classes; annotations such as “`@Persistent` ” and “`@Indexable` ” are used to declare specific database-related properties for these classes and data elements.
    3. Importing the `com.mcobject.extremedb.*` package enables developers to import classes such as *Database, Connection* and *Cursor* and use them to perform database operations.
    4. The interface to the *e*X*treme*DB runtime libraries is implemented through a set of dynamic link (Windows) or shared (*nix) libraries located in the subdirectory target/bin.so

## Using the C/C++ API

The basic steps in building an *eXtreme*DB C/C++ application are illustrated in the following diagram; the host components are in blue, while the target components are in red:

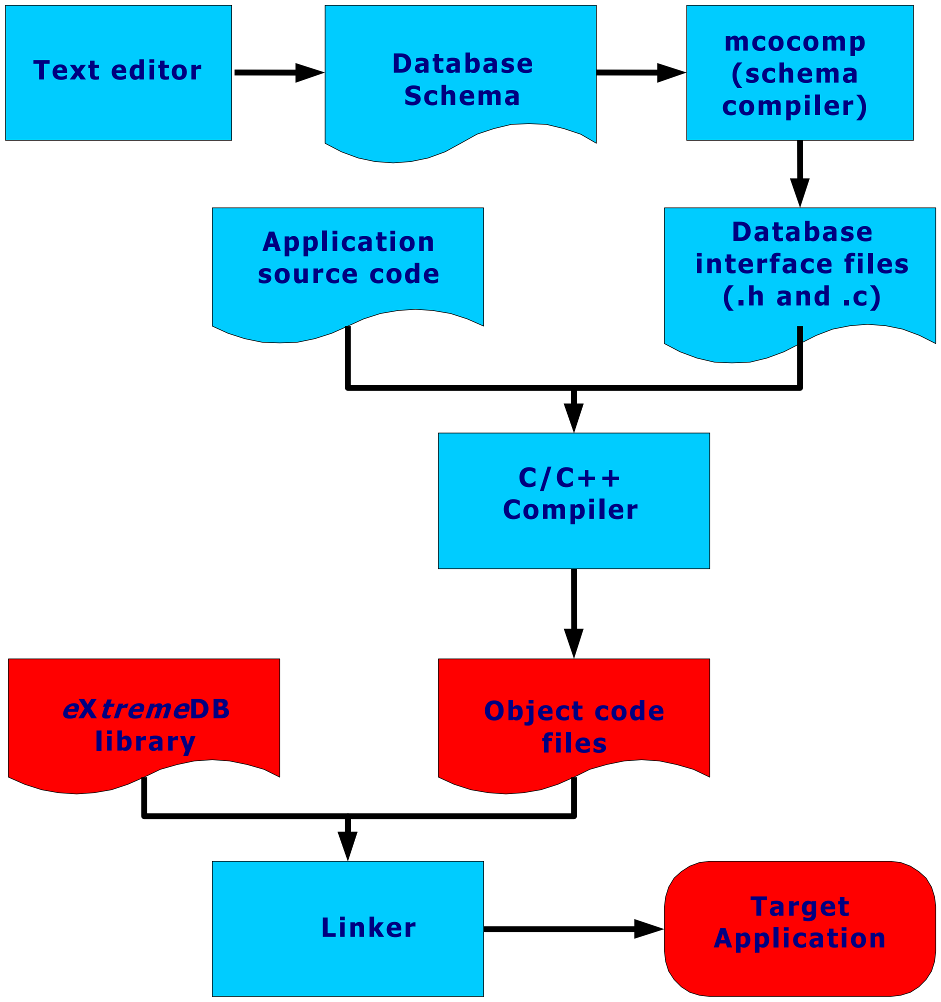

**Figure 1**

Suppose you have installed your package in directory `c:/eXtremeDB` (the *eXtremeDB* root directory) and we start with a simple “Hello world!” program like the following `main.c` in a new directory, samples/core/hello:

```
int main( int argc, char ** argv ) {

    printf ( "Hello eXtremeDB world!\n" );

    return 0;
}
```

Create a minimal database schema with a single class *MyClass* in the file `schema.mco`:

```
declare database hellodb;

class MyClass {
    unsigned<4> id;
};
```

Compile the schema with the command line:

```
../../host/bin/mcocomp schema.mco
```

The schema compiler produces two files, `hellodb.h` and `hellodb.c`, that contain the generated database dictionary and database access API necessary to access our simple database. To compile and link the source files `main.c` and `hellodb.c` some *eXtremeDB* runtime libraries must be added to the linker input list. The `target/bin` directory contains the *eXtremeDB* binary runtime libraries for the target platform. The combination of libraries linked into an application will depend on the specific *eXtremeDB* features the application will be using. (These libraries are provided in both “release” and “debug” versions and are explained in detail in the *release_notes*.)

At a minimum, a simple “Hello world!”-type program that creates an *eXtremeDB* database in conventional memory will require the following libraries (by convention we will use, e.g., “mcolib” which will exist as mcolib.lib on Windows platforms and libmcolib.a on *nix platforms):

| | |
|---|---|
| mcolib | “core” runtime library |
| mcovtmem.lib | “virtual tables” library for all-in-memory databases |
| mcomconv | “memory devices” library for conventional memory |
| mcotmursiw | “transaction manager” library for the MURSIW transaction manager |
| mcos[platform]n, where ‘platform’ refers to the target platform. For example “w32” for Windows (mcosw32n.lib), “lnx” for Linux | “synchronization” library |

| | |
|---|---|
| (libmcolnxp.a), etc. Refer to the *release_notes* for detailed information on library names) | |
| mcouwrt | “utilities” library |

Other configurations (e.g. shared memory instead of conventional memory, or MVCC transaction manager instead of MURSIW) require a different set of runtime libraries. These configurations and the corresponding libraries are described in detail in the *release_notes* installed in the root installation directory. Normally all necessary runtime components are specified in a makefile on Unix systems, or a project file if an IDE is used for development (Microsoft Visual Studio, WindRiver Tornado, etc.). In fact, all of the *e*X*treme*DB samples come with such a makefile or project file. These samples are presented in a sequence designed to help teach the fundamentals of *e*X*treme*DB quickly and thoroughly. Though not all the samples may relate directly to your project, all *e*X*treme*DB developers can find helpful information in the example code.

## Step 0: Schema Compilation

The starting point for all C/C++ projects is to create and compile the database schema as demonstrated in sample `00_ddl`. To build this sample, run “make” on Unix systems or open the project file in your IDE and build it. Note that the first step of the build is to invoke the *mcocomp* schema compiler. This produces the output files `sampleddl.h` and `sampleddl.c` that *must* be compiled and linked with the source files `main.c` and `common.c`.

Running `00_ddl` results in a single line of output. In fact, the topic of interest here is the file `schema.mco` that defines a database with multiple classes. It is instructive to examine this schema definition while referring to the *e*X*treme*DB *User Guide* for explanations of the different DDL statements, as well as the output files `sampleddl.h` and `sampleddl.c` produced by *mcocomp*. Note how specific type definitions and function prototypes and implementations are generated for each class defined in the schema.

These generated files should never be modified by hand; they will be replaced in their entirety if/when the schema file is modified and recompiled. Instead, put any application-specific code (e.g. #define or type definitions) in a header file and #include it in the schema file.

## Step 1: Runtime information

It is useful to examine *e*X*treme*DB runtime information as demonstrated in sample `01_rtconfig_inmem`. Build and run this sample and note the information displayed. In the source file `main.c` notice that a number of “sample helper” functions are called: `sample_os_initialize()`, `sample_header()`, `sample_show_runtime_info()`, `sample_pause_end()` and `sample_os_shutdown()`. These helper functions are implemented in file `common.c` and encapsulate many standard operations, allowing simplification of the sample application code to illustrate only the points of interest. Here it is instructive to examine functions `sample_header()` and `sample_show_runtime_info()` to see how runtime information is retrieved from the `mco_runtime_info_t` structure returned by calling function `mco_get_runtime_info()`.

If you are working with *e*X*treme*DB *Fusion*, another sample, `01_rtconfig_mixed()`, will be present in subdirectory `01-rtconfig/mixed`. This sample is identical to `01_rtconfig_inmem` except for the schema definition and the linker directive. Notice the declaration of class B as "persistent". This makes the database a "hybrid" database that includes both in-memory and persistent (stored on disk or flash memory) objects. The necessary persistent storage functionality is implemented by the *e*X*treme*DB "disk manager" and exported to applications through the "virtual table" library *mcovtdsk* and the file system-dependent "wrapper" library (such as `libmcofuni.a` or `mcofw32.lib`) as explained in the *release_notes*. Note the difference in the linker input directive between these two projects and the difference in the runtime information that results from the different library combination.

## Step 2: Opening databases

Before performing database storage and retrieval operations, we must open the database, as demonstrated in sample `02_open_conv`. *e*X*treme*DB uses a notion of "devices" to describe storage locations. For an all-in-memory database, there can be a single device to describe the conventional/local or shared memory for the database. For an *e*X*treme*DB Fusion database, there can be four or more devices to describe the in-memory portion (which is always present for the database meta-data), memory for the cache, the file storage for persistent data, and file storage for the transaction log. The `02_open_conv` sample demonstrates how to open an all-in-memory database in "conventional" memory (aka local memory), while sample `02_open_dbextend` demonstrates the use of an array of devices to allow the application to extend the in-memory database.

Sample `02_open_shared` illustrates how to open a shared memory database. The platform-dependent shared memory library is linked into `02_open_shared` rather than the conventional memory library *mcomconv*. Note that if you change the linker directive by substituting *mcomconv* the database open `mco_db_open_dev()` fails with an invalid parameter. This is because the share memory device `dev.type = MCO_MEMORY_NAMED` and parameter and `dev.named.name` are valid only for shared memory, which is implemented in the *mcomipc* (Linux/SunOS and HPUX), *mcompsx* (for OSes with the POSIX API such as QNX or VxWorks) or *mcomw32* (Win32) library – not with *mcomconv*. (See the *release_notes* for a more detailed explanation of the *e*X*treme*DB runtime libraries.)

Remember that when a database has been opened, the application should close it by calling `mco_db_close()` before terminating.

If you are using *e*X*treme*DB *Fusion* an additional set of samples – `02_open_disk_file(), 02_open_disk_multifile(), 02_open_disk_raid(),` 02_open_disk_hybrid(), 02_open_security disk_cipher() and 02_open_security_disk_crc() – will be present in subdirectory 02-open. These sample applications demonstrate how to define the devices array to open disk-based databases in the various forms their names indicate. Note that, while the data (or at least some of the data) for these databases is stored on a file system, the first two storage devices in the array are memory devices used for the "DATABASE" (and database meta-data ) and "CACHE" , while the database storage device and transaction log are of type "MCO_MEMORY_FILE".

## Step 3: Connecting databases

The next step after opening a database is to create a "connection" to it as demonstrated in the 03_connect_* samples. Sample 03_connect_single_task demonstrates how, once the database is successfully opened, the function mco_db_connect() is called to create the database connection. Once a connection is successfully established, the connection handle can be passed to various application functions within the same thread to perform database operations. When completed, the function mco_db_disconnect() should always be called to close the connection before closing the database and terminating the application.

Sample 03_connect_multi_task demonstrates how a handle to an opened database can be used by multiple threads to create *separate* connections to the same database. Sample 03_connect_multi_process demonstrates how to detect if a database is already open by first calling mco_db_connect(). If the error code MCO_E_NOINSTANCE is returned, then the database has not yet been opened, so the application opens it and connects. Otherwise, the initial call to mco_db_connect() creates the connection and the application proceeds as desired. To see the sample in operation requires running the first process and then starting a second instance of the program (using a separate console).

## Step 4: Database operations

Once the database is opened and a connection established, the application can begin performing read and write storage operations as demonstrated in the 04_operations sample. To perform any storage operation, the application *must* first start a database transaction, as described in chapter 4 of the *User Guide*. Once a transaction is opened, the transaction handle is passed to the generated _new() function to create a new instance of the object in the database. The object's handle is used for the generated _get() and _put() operations to insert values into the object. The new values are stored in the database when mco_trans_commit() is called with the transaction handle.

## Step 5: Indexes and cursors

As explained in chapter 4 of the *User Guide*, indexes provide for fast lookup and/or sorted retrieval of database objects based on some *search criteria*, and cursors provide the means for iterating through the objects. The 05_indexes* samples demonstrate *e*X*treme*DB's powerful variety of supported indexes. B-tree indexes are common to most database systems but many of the *e*X*treme*DB indexes enable specialized capabilities, such as handling geospatial data (R-trees) or multi-dimensional data (KD-trees).

## Step 6: Error handling

Chapter 6 of the *User Guide* explains the three categories of runtime return codes: status codes, non-fatal error codes and fatal error codes. The sample `06_errorhandling_statuscode` demonstrates a status return code and how applications check the status of various runtime function calls. The sample emphasizes the fact that status codes like `MCO_S_NOTFOUND`, `MCO_S_DUPLICATE` and `MCO_S_CURSOR_END` may be part of normal operations. Non-fatal error conditions are demonstrated in `06_errorhandling_nonfatalerr`. In contrast, *fatal* error conditions require special handling and are likely to indicate implementation errors, as shown in sample `06_errorhandling_fatalerr`. It is common practice to register a fatal error handler, as seen in prior samples, to halt the application if a severe runtime error is encountered. This is demonstrated in samples `06_errorhandling_fatalerr` and `06_errorhandling_fatalerrex`.

## Step 7: Transaction Managers

The section on Concurrency Management in chapter 4 of the *User Guide* explains the differences between the three transaction managers. Briefly, the **M**ulti-**R**eaders-**SI**ngle-**W**riter (MURSIW) transaction manager uses lightweight locks (called latches) to serialize `READ_WRITE` transactions while allowing `READ_ONLY` transactions to run simultaneously. The **M**ulti-**V**ersioning **C**oncurrency **C**ontrol (MVCC) transaction manager implements an "optimistic" strategy that can be tuned by setting transaction isolation levels. The Exclusive transaction manager is an alternative that provides optimal transaction throughput for single-threaded database access.

To demonstrate their different behaviors, the `07_transactions*` samples use a multi-threaded application that writes to a single database from different threads.

## The Samples document

The samples discussed above demonstrate basic functionality that nearly all *e*X*treme*DB applications will employ. However, there are many groups of samples in the SDK and some of these may not relate to your project. To help you find relevant samples, the *Samples* document (located in the /samples directory)  gives a brief description of each one. With the experience gained so far, you should be able to build and experiment with the examples that interest you.

## McObject Support

If you encounter problems, please feel free to send questions to support@mcobject.com. To facilitate quick and accurate responses we ask that you include specific code demonstrating your problem (where possible), the exact *e*X*treme*DB package you are using, and the relevant specifics of your development environment and target system.